1995-03

Integration of an image hardware/software system into an autonomous robot

Kisor, John Carl.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

INTEGRATION OF AN IMAGE HARDWARE/SOFTWARE SYSTEM INTO AN AUTONOMOUS ROBOT

by

John Carl Kisor

March 1995

Thesis Advisor: Yutaka Kanayama

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1995	Master's Thesis

4. TITLE AND SUBTITLE
INTEGRATION OF AN IMAGE HARDWARE/SOFTWARE SYSTEM INTO AN AUTONOMOUS ROBOT

5. FUNDING NUMBERS

6. AUTHOR(S)
John C Kisor

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The major problem addressed by this research is how to integrate an image understanding subsystem into an autonomous mobile robot so that the robot will be self-contained and independent of any unix workstation for extracting image information. The resulting image understanding subsystem should be a part of the total intelligent autonmous robot and should provide functionality that will allow the robot to determine its position and that of obstacles in a partially known environment. The image understanding subsystem should be fully integrated with existing motion control and sonar software.

The approach taken was to develop software to capture images that uses a VME bus to interface with a standard image manager. The software was written to provide a clean interface between the image grabber hardware and the robots existing Model-based Mobile Robot Language. By maintaining functionality similar to that provided in previous image understanding software, the changes needed to incorporate previous research software is kept to a minimum.

The result is a autonomous robot that can capture images using a standard image manager, display those images and then convert them to a format needed by existing image understanding software to extract information for position determination or avoid obstacles. The image understanding software uses only the computing power it has available on-board the robot. This gives the robot the capability to extract image information about its environment and remain autonomous and self-contained.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Robotics, Computer Vision, Image Understanding	80
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited

INTEGRATION OF AN IMAGE HARDWARE/SOFTWARE SYSTEM INTO AN AUTONOMOUS ROBOT

John Carl Kisor
Lieutenant, United States Navy
B.S., North Dakota State University, 1986

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

March 1995

Author: [redacted]

John Carl Kisor

Approved By: [redacted]

Yutaka Kanayama, Thesis Advisor

[redacted]

Chin-Hwa Lee, Second Reader

[redacted]

Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The major problem addressed by this research is how to integrate an image understanding subsystem into an autonomous mobile robot so that the robot will be self-contained and independent of any unix workstation for extracting image information. The resulting image understanding subsystem should be a part of the total intelligent autonmous robot and should provide functionality that will allow the robot to determine its position and that of obstacles in a partially known environment. The image understanding subsystem should be fully integrated with existing motion control and sonar software.

The approach taken was to develop software to capture images that uses a VME bus to interface with a standard image manager. The software was written to provide a clean interface between the image grabber hardware and the robots existing Model-based Mobile Robot Language. By maintaining functionality similar to that provided in previous image understanding software, the changes needed to incorporate previous research software is kept to a minimum.

The result is a autonomous robot that can capture images using a standard image manager, display those images and then convert them to a format needed by existing image understanding software to extract information for position determination or avoid obstacles. The image understanding software uses only the computing power it has available on-board the robot. This gives the robot the capability to extract image information about its environment and remain autonomous and self-contained.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

A. BACKGROUND

Yamabico is an autonomous mobile robot that has been used by the Naval Postgraduate School (NPS) as a platform for developing algorithms for motion planning, sonar return interpretation and image understanding. In previous image understanding research, all images retrieved by *Yamabico*'s camera have been processed on a Silicon Graphics Personal Iris™ workstation with the hope that future developments in hardware would permit the software's use on board *Yamabico*. Given the increased availability of the image hardware and greater CPU power, research can now be done using on-board computer vision. This research explores the development of software for Image Understanding with an autonomous mobile robot. *Yamabico* is an excellent platform for this research since its program can be altered, downloaded and tested in a matter of minutes. Many other research platforms require several personnel and hours of planning to test minor changes to a program.

B. OVERVIEW

Yamabico has been the focus of Dr. Yutaka Kanayama's research at NPS. The robot was designed to operate autonomously. It can move at speeds up to 65 cm per second both forward and backward. Since it is has two drive wheels, *Yamabico* can rotate on axis. Four caster wheels are located at the four corners of the robot to provide stability. It maintains a precise knowledge of its location through dead reckoning (DR). It is *Yamabico*'s smooth motion [Kanayama 94] that allows the robot to DR so accurately. *Yamabico* is shown in Figure 1.

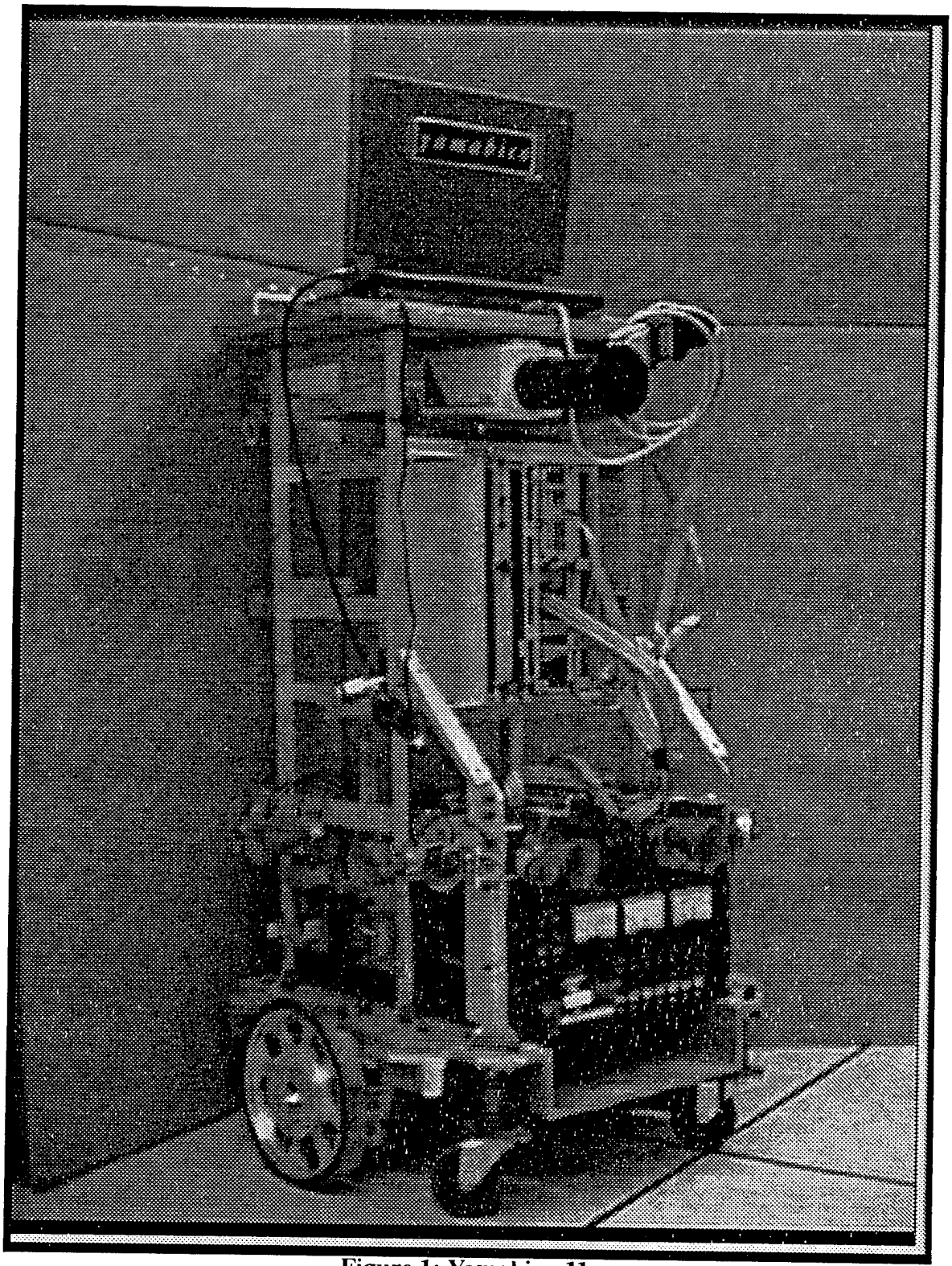

Figure 1: Yamabico-11

C. YAMABICO-11 - AUTONOMOUS MOBILE ROBOT

Yamabico-11 is designed to be a fully autonomous robot. All its hardware is housed in an aluminum frame which is 25 inches square at its base and 36 inches tall. Electrical power is supplied by two 12-volt motorcycle type batteries. In the summer of 1994 the Motorola 68020 CPU and 68881 floating point processor were replaced with a Ironic's SPARC-4 processor. Processor differences are shown in Table 1.

Processor	68020	SPARC-4
CPU RAM	1MByte	16MByte
Clock Speed	12.5 MHz	33MHz
RAM Card	4MByte	none

Table 1: SPARC-4/68020 Comparison

Having the SPARC CPU on the robot allows code to be developed at any of NPS's Sun SPARC workstations. Currently the code is compiled using the GNU C compiler. Communication with the robot is through a telnet connection. A standard loader combines object programs to create an executable file. The Ironic's bootstrap program "bootp" actually loads the program into memory at the address specified by the executable. The program is downloaded using the Trivial File Transfer Protocol (TFTP). Programs are executed by the "run" command which, along with bootp, is resident in the Ironics ROM. The ease of developing code for *Yamabico* is clearly one of its advantages.

Although the actual execution of test programs on *Yamabico* is very simple and the high level user commands are easy to understand and use, the development of the software to support those commands is much more challenging. This is because some of the conveniences that can be taken for granted in a workstation are either not found or more primitive on *Yamabico*. Some of these limitations are summarized in Table 2.

Capability	Workstation	Yamabico
Memory Management	Memory allocation routines are contained in operating system. Safeguards in place to control allocations. Virtual memory often used to augment actual memory. Systems are usually forgiving, if garbage collection not performed.	User must be conscious of memory limitations and garbage collection
Disk Storage	Complex and complete systems that often augment actual memory	None available
Software	Plethora of software available for many applications including image processing	All software must be written to interface with Yamabico's operating system

Table 2: Workstation/*Yamabico* Comparison

Control information for moving the robot is first sent from the SPARC CPU to the dual axis controller board and two 4-channel serial communications boards through a VME bus. The dual axis controllers interface with driving and braking motors. A conceptional diagram of the entire hardware system is shown in Figure 2. Information on *Yamabico*'s configuration is maintained and includes an x,y position, current heading of the robot (theta) and the magnitude of turn (kappa). Kappa is critical to Yamabico's smooth motion [Kanayama 94]. *Yamabico*'s position is updated through dead-reckoning, although several attempts to improve this with sonar [Lochner 94] and image understanding [Peterson 92, Stein 92] have been made with limited success.

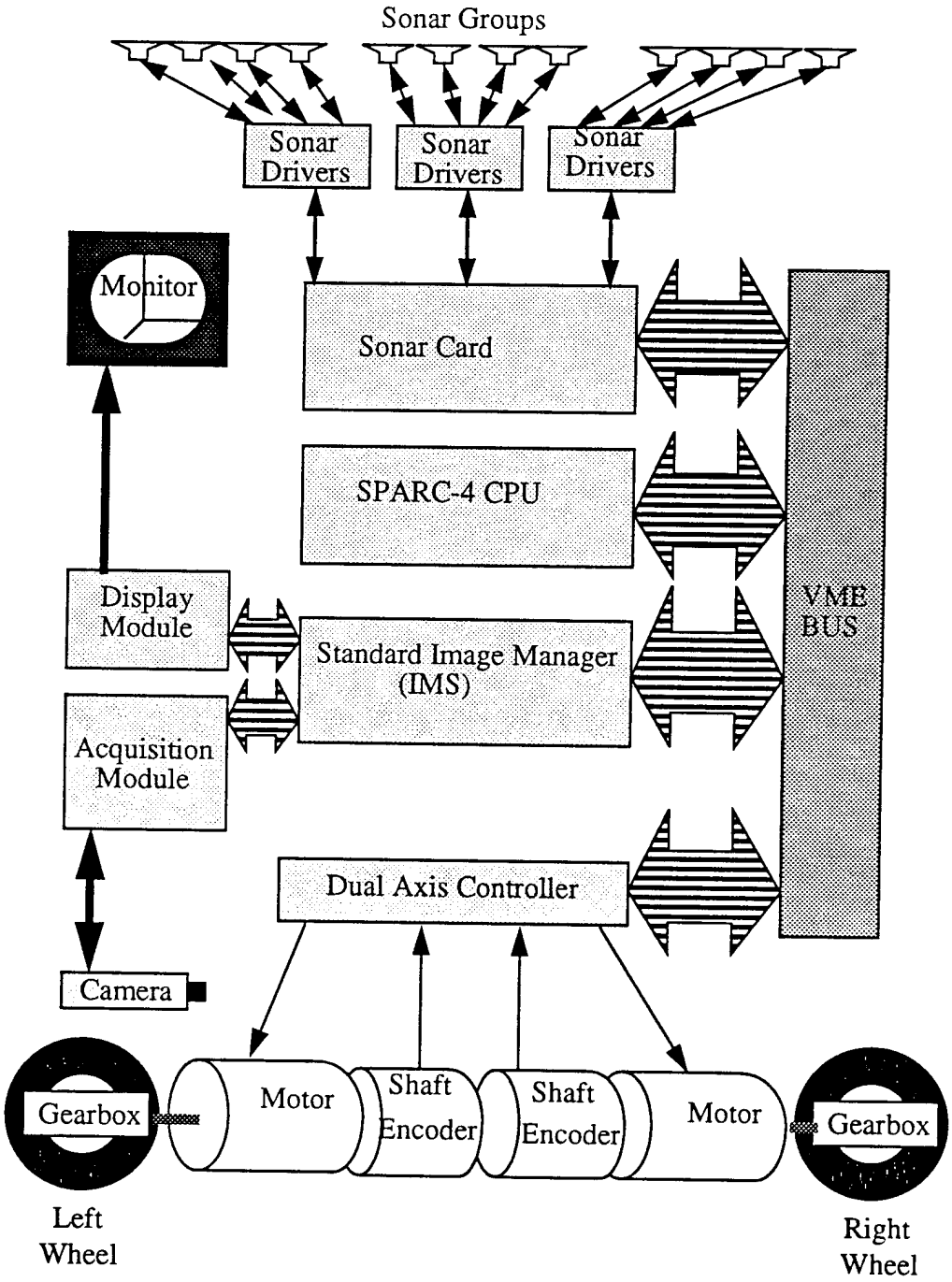

Figure 2: Yamabico Control Architecture

Yamabico has 12 sonars which provide sonar data for 360 degrees of azimuth and out to ranges of approximately 400 centimeters (cm). The sonars are located 40 cm from the ground and radiate horizontally at 30 degree intervals from the robots center. The sonars have been used extensively to maintain wall and large object clearance. There have also been experiments in position determination using *Yamabico*'s sonars.

D. MODEL-BASED MOBILE-ROBOT LANGUAGE (MML)

Central in the development of *Yamabico* has been the Model-Based Mobile-Robot Language (MML). The purpose of this language is to provide the robot user with commands that would be useful in producing desired behaviors. The long-term goal is to have commands such as gotoGoal(configuration) which would result in the robot traversing its environment to end up in the desired configuration. Presently there is no such path planning, but the commands to support this operation are being developed. MML currently has commands that direct the robot to follow lines and circles. *Yamabico's* motion control subsystem directs the robot through smooth transitions between those lines and circles. There are also commands that allow the robot to wall-follow, and the rudimentary commands exist for object avoidance using the sonar.

Most recently, MML has undergone a reorganization that took place during the transition from the Motorola 68020 CPU to the SPARC-4. This redesign of the MML software system focused on creating independent software sub-systems, the encapsulation of data, reducing hardware dependencies and minimizing assembly code. The result is that MML is easier to understand, modify and is more portable to other hardware platforms.

E. MOTIVATION FOR VISION DEVELOPMENT

1. Image Understanding

Computers have been able to acquire and store images in a digital format for more than twenty years. The motivation to extract information from images is probably rooted in our own (human) dependence upon this sensor. Image understanding encompasses many

areas from object recognition, including facial recognition, to object orientation determination in manufacturing, to our own work in environment recognition and conversely obstacle detection. Since image understanding can mean many things, it is important to state what it means to the Yamabico group.

Image Understanding: Use of computer vision to extract information about the environment in which the camera exists.

Research at NPS has focused on extracting lines from the image. Recently, there has been an emergence of special hardware to accomplish this task. The information that Yamabico is able to extract from its environment is simplified somewhat by some environmental assumptions:

- Operation is in an indoor and consequently orthogonal environment
- The environment is only traversed in the horizontal plane
- A partial model of this environment is available

2. Autonomous Robot Vision

The motivation for developing an autonomous robot is to have that robot take over or augment some of the more mundane, repetitive or dangerous tasks that humans may be required to perform, This could be delivering ordinary mail in an office building or secret messages in the pentagon. The biggest advantage to using a robot for these types of tasks is that it frees humans to do the tasks that require a higher degree of intelligence. With budget shortfalls such as those recently felt by the Department of Defense it may be a cost effective way to free humans from those routine tasks and thus allow personnel to become competent at more demanding tasks. Also, there are tasks which we may not want humans to perform due to the risk involved, such as the inspection of hazardous areas.

3. Image vs. Sonar

The need for a image sensor is better seen when its capabilities are compared to those of the sonar. These are summarized in Table 3. Clearly, no one sensor has complete advantage over another. By adding an image understanding capability, the sensing

capability of the whole system should be improved. The greatest advantage of image processing over sonar is its range and its ability to sense objects that consist of a soft material such as the pant legs of a person. Other objects that may be sensed more reliably by an image based sensor are round or irregular objects.

Sonar	Image Sensor
Range up to 400 cm	Line-of-sight
Very little computation needed	Very computation intensive
Range information is easily obtained	Range information is more difficult to obtain and less accurate.
Range to nearest object given with each sonar ping	Two dimensional array of light intensities given for each image
Strength of sonar ping return based on material encountered	Strength of return based on ambient light
Sonar return can get interference from extraneous noise	Reflections can produce false objects and lines

Table 3: Sonar / Vision Comparison

F. VISION BASED SENSOR SYSTEM

The first work in developing an image understanding system for Yamabico was done by Kevin Peterson [Peterson 92], Jim Stein [Stein 92] and Mark DeClue [DeClue 93] The result of their work was the development of software that could extract edges from an image and match those edges to a known world model.

Although a Silicon Graphics Personal Iris^{TM} was used for development of the software and only the camera was located on the robot, the software was developed with the idea that it would eventually become part of an on-board system. The camera used was a JVC model TK870U which produced a RGB formatted image with 8 bits each for red, green and blue components. Images were captured using a Silicon Graphics video framer.

This system is ideal for development since its powerful graphics capabilities allow the programmer to display the results of image computations graphically. Lines extracted from the image can be displayed graphically, separate from the corresponding image or superimposed onto the picture.

All code was written using the ANSI C language. Peterson's work develops the algorithms for edge extraction using the sobel operator and linear fitting using least squares fit. Stein's work involved developing a 3 dimensional model that could be used to model orthogonal environments in which *Yamabico* can exist.

G. PROBLEM STATEMENT

The major problem addressed by this research is how to integrate an image understanding subsystem into an autonomous mobile robot so that the robot will be self-contained and independent of any unix workstation for extracting image information. The resulting image understanding subsystem should be a part of the total intelligent autonomous robot and should provide functionality that will allow the robot to determine its position and that of obstacles in a partially known environment. The image understanding subsystem should be fully integrated with existing motion control and sonar software.

The resulting software should be integrated into *Yamabico*'s Model-based Mobile Robot Language and have functionality similar to that needed by previous image understanding software. This will minimize changes to the existing software originally written for the Iris workstations.

H. ORGANIZATION OF THESIS

This thesis is organized around the steps taken to construct the low-level image software routines. The following chapter describes the hardware used for the new image subsystem, including the standard image manager, acquisition module, display module and the CPU interface. Chapter III discusses the image functionality that is needed to integrate image understanding software into MML. That functionality and how it was implemented

is described in Chapter IV. The fifth chapter describes problems associated with executing image software routines along with the motion control and sonar software. Chapter VI describes how the implementation overcame the problems discussed in Chapter V. The results of this work are shown in Chapter VII. Chapter VIII concludes with a summary and recommendations for future research.

II. IMAGE HARDWARE DESCRIPTION

A. LOCAL BUS STRUCTURE

Yamabico's IV-SPRC-xxA CPU board has 16Mbytes of DRAM, located in the lower portion of the 4Gbyte address space. The address space above this 16Mbytes is devoted to VME bus use. It contains several regions as shown in Figure 3. The address space above these VMEbus regions is reserved for EPROM and board configuration registers.There are three logical mappings to the VME bus: the cluster-internal virtual bus, the synchronous Mbus and the asynchronous T-bus. All of *Yamabico's* address space has been mapped using the T-bus.

The VMEbus interface which resides on the T-bus allows operations with Motorola 680x0 type protocols. A T-bus master may only access other T-bus devices and local DRAM. T-bus features are:

- •Fully asynchronous bus
- •Separate 32-bit address and data buses
- •Four Gigabytes of physical memory address space

The T-bus 32-bit address space is conceptually divided into seven regions. Some of these regions are fixed and others have programmable sizes. Figure 3 shows the default address space which was used in this system. The space is initialized by resident initialization code in the Ironics SPARC CPU card. There are two regions which involve the Yamabico's image subsystem: Region 3 and A24 space.

1. Region 3 Address Space

Region 3 starts at the end of region 2 and extends to the bottom of the EPROM address space (0xff000000). The attributes of this region are set in the Region 3 Attributes Register (0xfffd0b00). The Ironics initialization code initially sets up this region for addressing and data which contain 32 bits (A32/D32). For *Yamabico*'s sonar system this

was switched to 16 bit addressing (A16) and 16 bits of data (D16). This is the same setup needed for Imaging Technology Inc.'s Standard Image Manager (IMS). To avoid conflicting with the address space used by the sonar, the image board can be offset from the base address of this region up to 0xff00 in 0x100 intervals. The default offset, 0x0600,

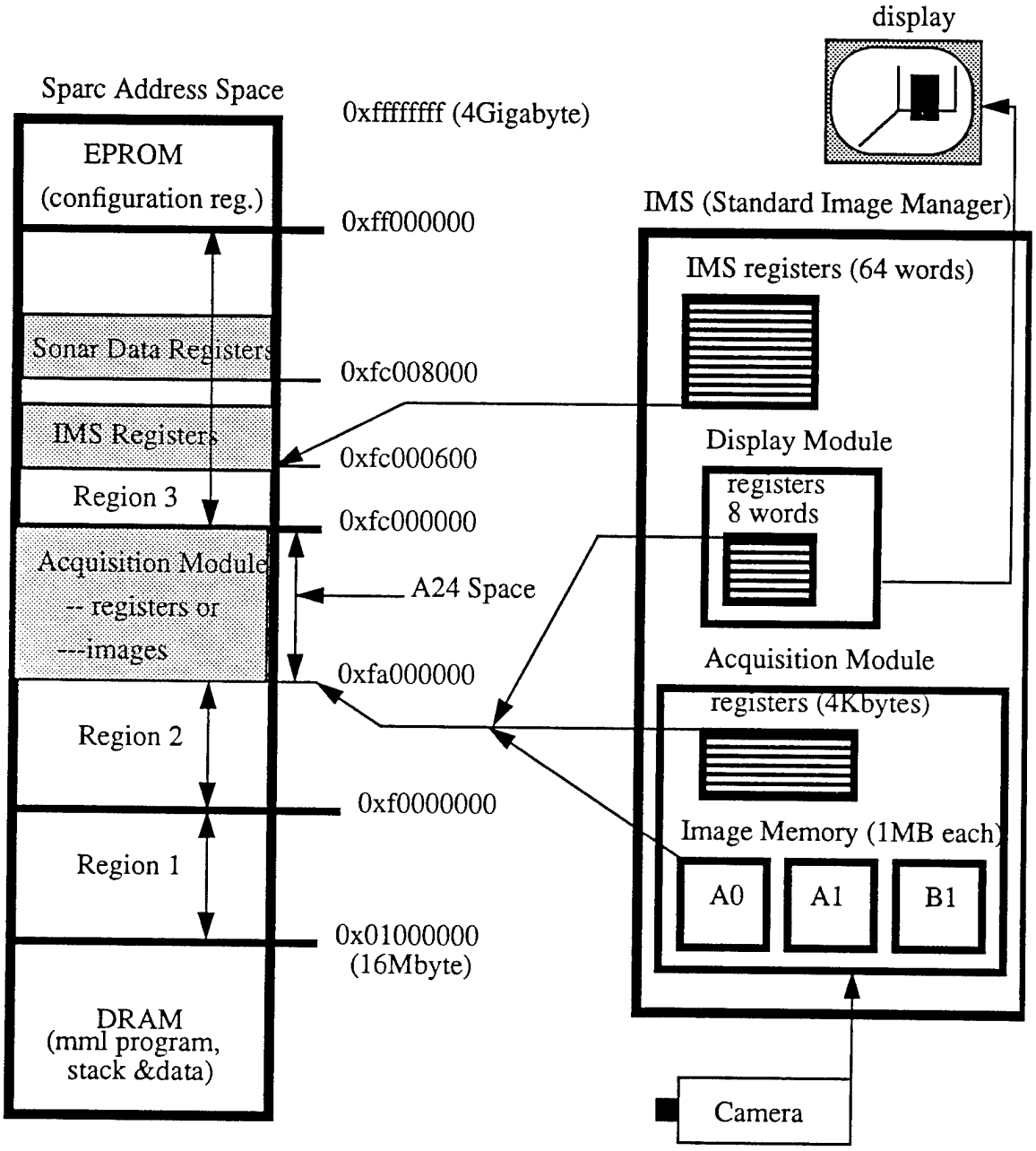

Figure 3: VME Mapping of Image Modules into SPARC-4 Address Space.

was chosen for *Yamabico* since it does not conflict with the VME address space for the sonar or the dual axis controller registers. The sonar registers are mapped to 0xfc008000 and the image board is mapped to 0xfc000600 and uses 64 words (0x00-0x7f).

The image board addresses in this region are used to set initial configuration of the image board. This includes setting up address space for the actual images and additional modules such as the acquisition module and the display module. Configuration information stored in this region includes:

- Size mapped into VME memory space (image memory)
- Address space desired (24 bit of 32 bit)
- Input frame masking bits
- Page selection (image, acquisition module, display module

There are three other contiguous regions above the actual DRAM memory address space.

2. Region 1 Address Space

This is the area supports the same three addressing modes as region 3: 16, 24 and 32 bit. It can deliver data as 16 or 32 bit blocks. The region starts at the end of local DRAM space and extends through the address specified in the Boundary 2 address register (0xfffd0600). The attributes of this region are set in the Region 1 Attributes register (0xfffd0900). The Ironics initialization code maps region 1 to the VMEbus as A32/D32.

3. Region 2 Address Space

This region supports the same addressing/data modes as region 1 and 3. It starts at the end of region 1 extending up to the start of region 3.

4. A24 Space

The address space chosen to contain the actual images plus the acquisition module registers and display module registers is called A24 space. This region can overlay all three regions previously discussed. It can start anywhere from the top of local DRAM up to the

bottom of EPROM address space. Yamabico's image subsystem uses the Ironics default for this region which is from 0xfa000000 to 0xfbffffff. The default setting also assumes a 16 bit data width for the slave board. This default setting is also used, although data can be up to 32 bits wide.

B. STANDARD IMAGE MANAGER (IMS)

1. Standard Image Manager Configuration

After setting up the A24 Space attributes some IMS configuration registers must be set to allow the acquisition module registers to map to this region [ITIIMS 93]. These include the IMS configuration register, and page select registers. The IMS status register can also be checked to confirm that the IMS board is present. The configuration register contains bits that set the amount of memory to be mapped to VME address space, selection of standard or extended addressing and the 3 high order bytes of an 8 byte address offset from the beginning of the region selected.

As shown in Figure 3, the acquisition module is mapped to 0xfa000000, which is the beginning of A24 space. Standard addressing (24 bits) was selected with a map size of 1MB. Module register access is not affected by this mapping size. With a map size of 1 Mbyte, the three images, the acquisition module registers and display module registers can be mapped to VME address space using the page register. The page register allows the setting of pixel size, choice of enabling or disabling VME memory access and the choice of selecting either one of three memory pages or the acquisition and display modules. Initially the page register is set to access the acquisition module registers through VME memory addresses. Figure 4 shows the flow of pixel data through the image manager.

2. IMS Registers

Since the IMS is made to support many different functions, there are a multitude of registers which need to be set prior to capturing of an image. The most significant of these are described below.

a. Configuration

This register configures the memory base addressing mode and the amount of image memory mapped into the VME memory address space. The options used are:

- Map Size: 1MB allows access to all three images through paging
- VME Addressing format: Standard addressing (24 bit addressing)
- Memory base address: Offset from the beginning of SPARC A24 space

b. Status Registers

A read-only register that allows the user to test for:

- Presence of add-on modules
- Camera status
- Bus status

c. Frame A Mask

This register allows the programmer to mask input into frames A0 and A1. This is most often set to all binary ones to allow the writing of all bits to the frame.

d. Constant/Frame B Mask

Same as the Frame A mask except for frame B1. It also sets a constant to be used by the cross-port switch. This controls the way data is sent over the IMS's internal buses.

e. AOI (Area of Interest) Registers (9)

These 9 registers control the area of a picture to be put on the bus. For instance a portion of the frame in A0 can be passed to B1. This allows a program to concentrate on a specific portion of an image. In later versions of *Yamabico*'s image

understanding system, this could be used to "blow-up" a portion of an image in which an obstacle was detected for further analysis.

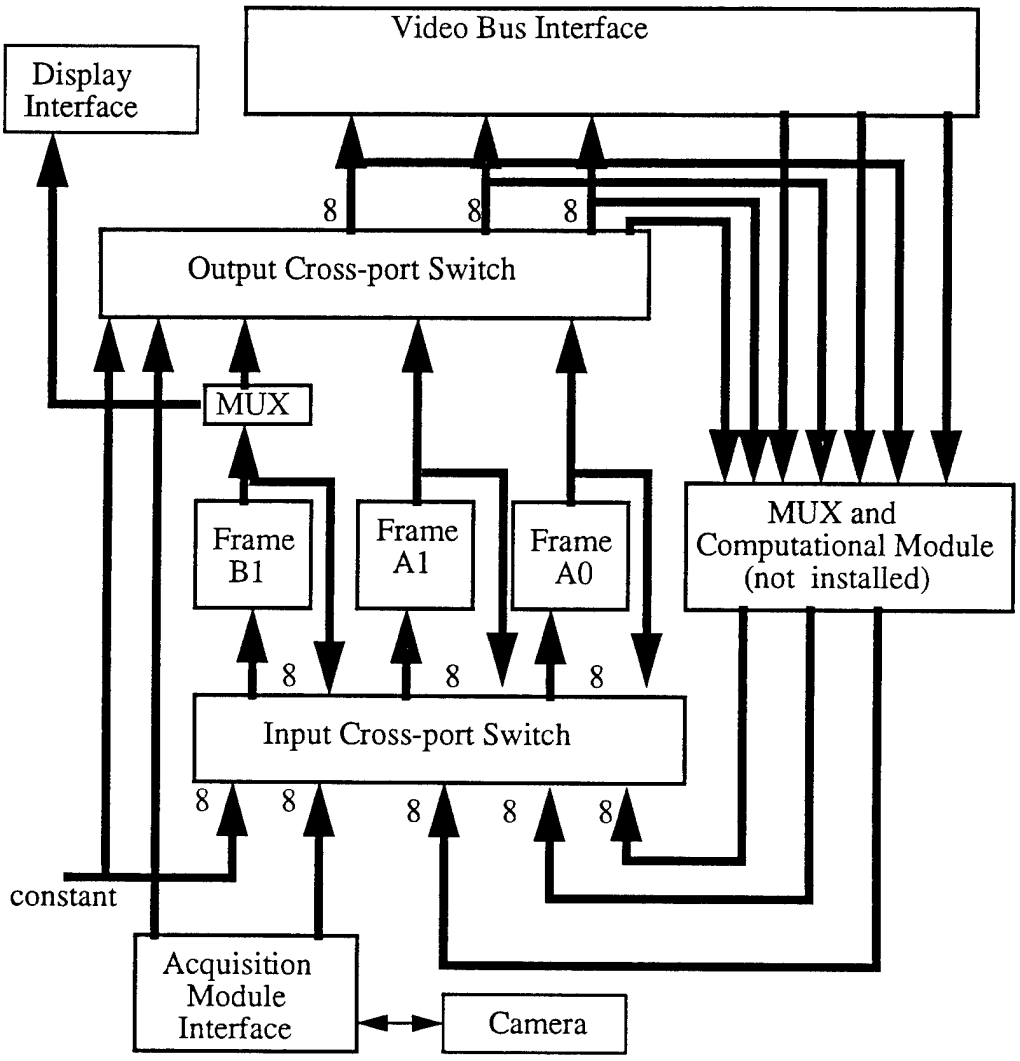

Figure 4: Standard Image Manager (IMS) After [ITIIMS 93].

f. Page

This register allows the programmer to specify which of the module registers or image frames is to be accessed. It also specifies the pixel size that should be expected, which in our case is 8 bits. The page register also enables VME access to the IMS image manager memory or modules

g. IMS Control

This register controls several aspects of the image board as a whole. It determines the role (slave or master) of the image board. Since we only have one IMS board this is always set to master. The IMS control register also:

- Determines the clock speed to be used
- Determines whether of not the display module will be enabled
- Sets the AM trigger mode

h. Interrupt Registers

There is no interrupt capability for *Yamabico*'s image system at this time; however, in later versions of the image subsystem this register could be used to allow the CPU to interrupt the image manager. This could allow *Yamabico* to capture an image based on events occurring in other subsystems such as the sonar, rather than as a part of a sequential program as is currently done.

i. Frame Acquisition

These registers consist of three sets of registers that control the way each frame acquires an image. They are very similar for frames A0, A1 and B1. Each contain the following bits

- **Trigger mode**: Determines whether the acquisition begins and ends upon detection of an external trigger pulse. The external trigger is supplied by the acquisition module.
- **Wait Acquisition Start**: If this bit is set, the acquisition is prevented from occurring until a global acquisition enable is received This allows frame operations to be synchronized.
- **Acquisition Mode**: These setup the IMS module to receive the image and

determines whether it will receive a single or multiple images. If the Wait Acquisition bit is not set, the image will be retrieved immediately otherwise it will be deferred until the acquisition enable register is written to.

j. Frame Scroll and Pan Registers

These are not currently used. They are set to zero so that no scroll or pan is produced.

k. Frame B Zoom Registers

This register is also not currently used by Yamabico, but it could be used to concentrate on a specific portion of the image.

l. Computation Registers

Yamabico does not currently have the hardware to support operations that this module performs. Although if this hardware was available, sobel operator calculations could be done in the computation registers instead of in software.

m. Frame Input Path Registers

This set of three registers allows each of the three frames to select the source of the image transfer. Images can come from the camera, another frame or be a constant value.

n. Cross-port Switch Control Register

This register configures the bus width and direction between the different frames, and the camera. It is set to a constant value.

C. ACQUISITION AND DISPLAY

The IMS board is supported by two smaller plug-in modules: the Acquisition Module and the Display module. The Acquisition module provides an interface between the camera and the IMS board. The display module is described in the next section and interfaces with a display monitor.

1. Acquisition Module (AM)

The acquisition module chosen for Yamabico is produced by the same company as the main image board (Imaging Technology Inc.)[ITIAM 93]. It is designed to interface with many different cameras and operate in a variety of modes. This is both an advantage and a hinderance since the 17 registers augmenting the 43 registers on the main image board must all be programmed. The module itself can receive up to 24 bits in parallel, but the camera chosen is only an 8 bit grayscale. Our 8 bit version uses a RS-422 for data input. The camera's 8 bit pixel data is first stored into a 4k by 8 bit FIFO queue and then transferred to the motherboard. Table 4 summarizes the main features of the acquisition module.

Model	AMDIG-8D
Max # of bits	8
Data Format	RS422
Timing Signals	TTL/RS422
AM-DIG Control Inputs	RS422
Max Clock Rate	20MHz

Table 4: Summary of Acquisition Module

Figure 5 shows the data flow from the camera and its associated timing signals. In the 8 bit version of ITI's acquisition module the 8 bit camera data is passed directly to a 4K by 8 bit FIFO queue. The 8 bits are duplicated on three output channels. Since our camera is an area scan device, the acquisition module outputs horizontal and vertical frame timing to the mother board based on the line enable, frame enable and pixel clock inputs from the camera.

a. Camera Timing Inputs

Camera timing inputs for our camera are RS422. They are shown on the left side of Figure 5. Line enable (LEN) is used to enable the valid video window generator horizontal offset and horizontal active counters. To support many different cameras, line enable can be triggered on the rising or falling edge. Since the line enable is inverted on Yamabico's camera, the falling edge of the camera's LEN is the clock for the horizontal counter. The frame enable input (FEN) is used to enable the valid video window generator vertical offset and vertical active counters. Since FEN is not inverted the rising edge of the

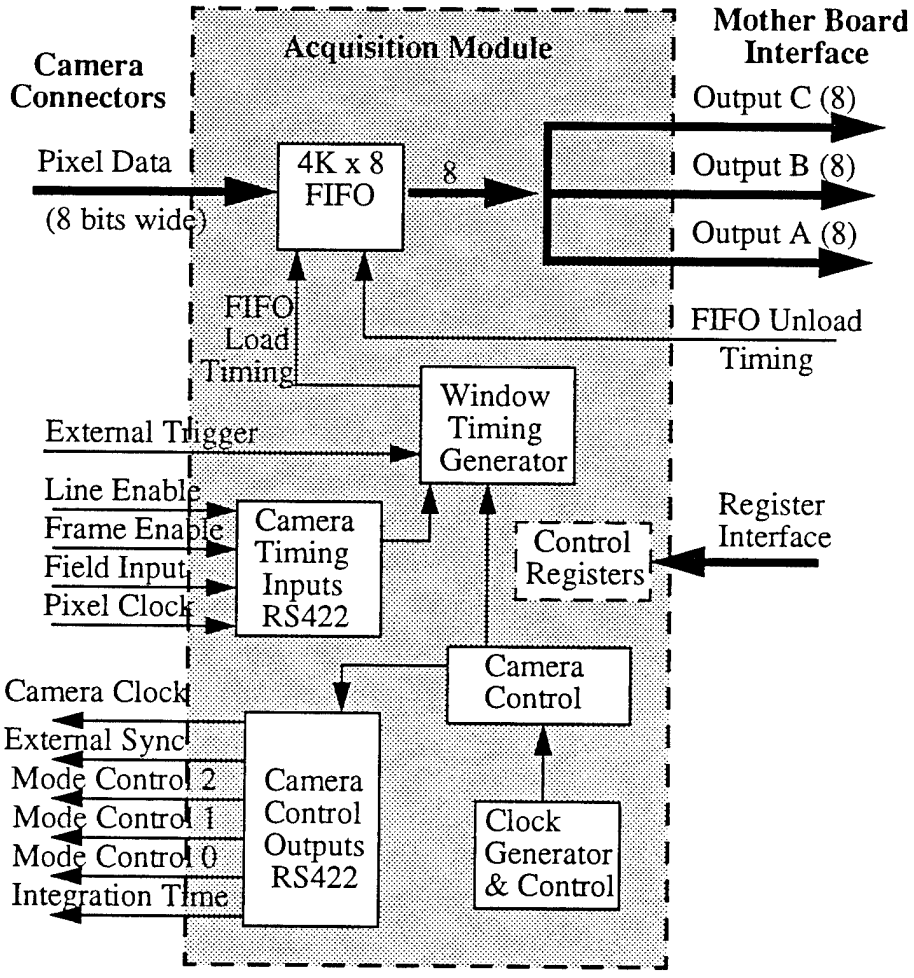

Figure 5: Acquisition Module Block Diagram After [ITIAM 93]

camera's FEN or Vertical Drive output enables and clocks the vertical counters. Pixel Clock Input (PCLK) is a data and control strobe for input. For *Yamabico's* camera all signals and data are sampled on the falling edge of the camera clock output. The external trigger input (EXTRIG) is used to synchronize the mother board video acquisition to an external event. For our camera the falling edge of the EXTRIG is used to trigger an external synchronization through the EXSYNC generator, that is, the falling edge of the EXTRIG causes an external trigger or starts the EXSYNC counter, if the trigger is enabled. The FIELD input line is used to control interlaced cameras such as the Cohu-4110. Since the clock output is tri-state whenever the AM is operating in interlaced mode, the AM cannot supply a clock to an interlaced camera.

b. Camera Timing Outputs

There are several output signals that are designed to support a variety of cameras. These are shown in the bottom left corner of Figure 5. The camera clock output (CCLK) is a programmable frequency that matches the frequency of the external clock. There are 12 standard frequencies that are selectable. The number of frequencies available is increased by the availability of eight clock divisors, which results in 96 different frequencies. Since our camera only supports an 8 bit wide data path, three of the 16 data pins are used for camera mode control. The external synchronization output signal controls camera timing and integration. There are four external synchronization modes:

- **Free running**: In this mode the EXSYNC will continuously output a pulse at a programmable length (retriggering itself).
- **External Trigger Mode:** EXSYNC pulse will be asserted for the programmed period of time whenever there is an external trigger.
- **Vertical Blanking Mode:** EXSYNC pulse will be asserted for the programmed period, whenever the acquisition module enters vertical blank.
- **Software Trigger:** EXSYNC pulse is asserted whenever the host writes to the Software Trigger register.

The EXSYNC counter can be disabled; thereby, generating integration or exposure times that are longer. EXSYNC can also be used as a frame reset.

c. External Trigger Modes

When operating with area scan cameras such as the Cohu-4110, the acquisition module initiates a mother board triggered acquisition at the start of the next full frame after receiving an external trigger pulse. The acquisition module will terminate the trigger cycle after the mother board has acquired one full frame.

d. Window Generator

The AM window generator allows acquisition of a full camera image or a sub-section of the camera image. Four registers (horizontal active, vertical active, horizontal offset, and vertical offset) control the position and size of the video window. The maximum window allowed by the acquisition module is 4K horizontal by 4K vertical, or 8K by 4K in interleaved pixel mode.

e. FIFO Data Buffer

The FIFO (first-in first-out) data buffer provides buffering for the transfer of pixel values between the camera and the mother board. The timing of the transfer of those bits is based upon camera timing inputs. There are five registers that control the size of a "valid video window" which is the block of transferred data. The window can be a portion of the entire image or the entire image. The mother board transfers pixel data at the end of every line.

f. Actual Registers

As stated above the acquisition registers control the interface between the camera and the image mother board which stores the images. Here is a detailed explanation of each register and how the registers are set up for *Yamabico*.

(1) **Camera Clock Control:** This register sets the output clock frequency to the camera. The clock frequency of *Yamabico's* camera is 14.32 MHz. A divisor for this frequency can also be set in this register; however for the Cohu 4110

camera, the divisor is not needed and therefore set to one. There are also three mode control bits supplied to the camera from this register.

(2) **Camera Timing Control:** This register sets the timing interface to the camera. These bits tell the acquisition module how to interpret signals coming from the camera. For example the Cohu 4110 samples each pixel on the falling edge of each clock cycle. The horizontal timing is enabled on the falling edge of the line enable (LEN) signal and vertical timing is enabled on the falling edge of the frame enable (FEN) signal. Since the Cohu 4110 is an interlaced camera, Interlaced Mode must be selected and another bit is needed to determine which portion of the interlaced image is selected. These are set with the Interlace Mode bit and the Field Polarity Bit which defines a low FIELD signal to represent an even field.

(3) **External Synchronization Control:** This register controls the external synchronization frequency, length and polarity. This is needed to trigger and synchronize the camera or external events and is often used to affect integration control and shutter control. The programmer has the option of making the synchronization occur continuously, upon the occurrence of an external trigger, with the occurrence of a software trigger or with the occurrence of a software blank.

(4) **External Synchronization Time:** The register controls the duration of the external synchronization pulse.

(5) **LUT Control and LUT Page Select:** Since look up tables are not needed for the 8 bit version of the acquisition module, this register is not used.

(6) **Horizontal Offset:** These bits determine the location of the first valid pixel of each line relative to the selected edge of the LEN. The horizontal offset counter is loaded with the selected edge of the LEN and decrements every pixel clock cycle. When the horizontal offset counter reaches zero, the horizontal active counter (see next section) is enabled and the acquisition module starts loading the FIFO. If horizontal offset

is zero the FIFO starts loading at the first clock edge after LEN. A graphical representation of the horizontal timing for the AM with the Cohu-4110 camera is shown in Figure 6.

(7) **Horizontal Active:** This register sets the number of valid pixels in each line. When the horizontal offset counter reaches zero, the acquisition module enables the horizontal active counter, and starts loading the FIFO. When the horizontal active counter reaches zero, the acquisition module stops loading the FIFO until the next line.

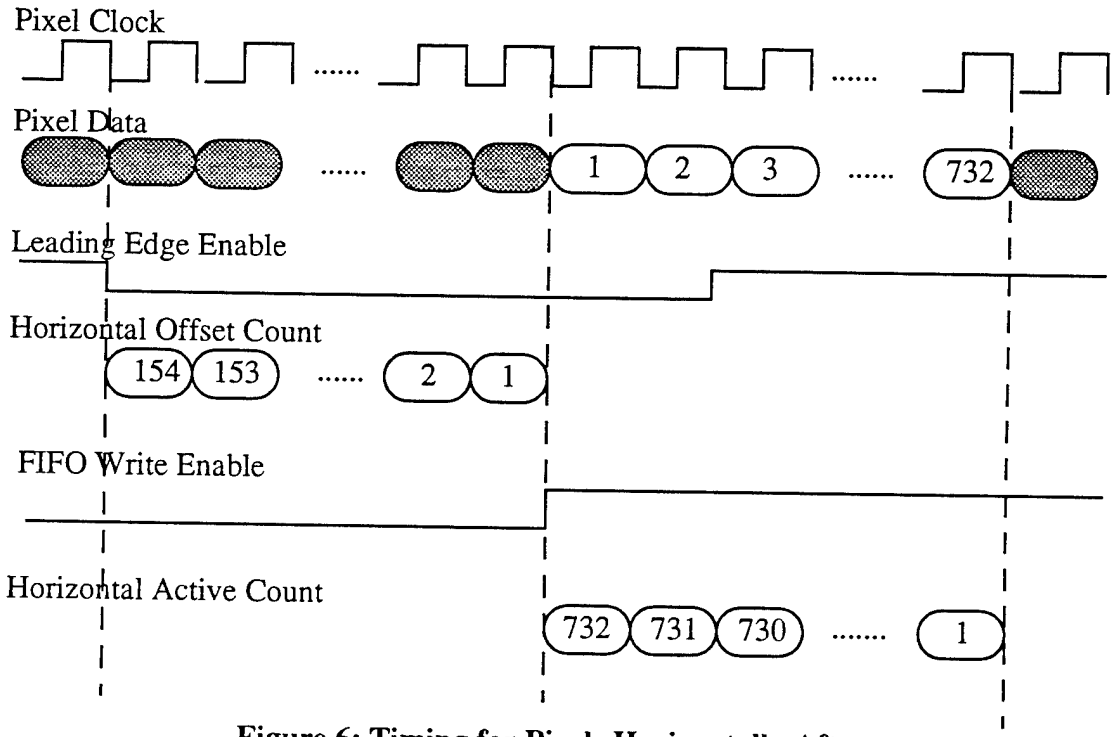

Figure 6: Timing for Pixels Horizontally After [ITIAM 93].

(8) **Vertical Offset:** This register sets the location of the first valid line of each frame relative to the selected edge of the frame enable input. The vertical offset counter is loaded with the selected edge of FEN and decrements every line cycle. When the

vertical offset counter reaches zero, the AM signals the mother board that the first valid line is ready.

(9) **Vertical Active:** This sets the number of lines in each frame. When the vertical offset register counter reaches zero, the AM starts unloading lines from the FIFO to the mother board and enables the vertical active counter. The vertical active counter decrements every line. When the vertical active counter reaches zero, the AM stops unloading valid lines to the mother board.

(10) **Trigger Control and Status:** This register enables the trigger if needed and sets the source of the external or internal trigger.

2. Display Module

The DM-PC Pseudocolor Display Module (DM) is a plug-module for the image manager. It provides a medium resolution pseudocolor RGB display for many types of monitors including 1024 by 768 non-interlaced and up to 1024 by 1024 interlaced monitors. It receives 8 bits of image data from the mother board (IMS module) and converts it into RGB pseudocolor. Overlay memory is supported for applications requiring graphics to overlay images. The heart of the display module is the Texas Instruments TMS34010 Graphics System Processor (GSP) which controls all graphics and image display functions. All display module registers are in-turn mapped to the GSP registers. The pseudocolor transformation is performed by a Bt478 RAM digital-to-analog converter (RAMDAC). All options for displaying are software programmable. The route of data through these components is shown in Figure 7.

The display module displays an image that is stored in frame B1 of the IMS mother board. Therefore it does not do any processing on the image, other than the mapping that is done to display the image in a RGB format. The DM does support an overlay which can be programmed to display menus or text. This feature may be used in later research to display lines generated by edge finding software.

The RAMDAC uses a look-up table (LUT) for the mapping of the 8 bit grayscale image on to a 24 bit RGB display image. There is also program memory available on the GSP which can be used to store operation code, such as the TIGA^{TM} graphical user interface. There are three main hardware components for converting eight bit digital data into an analog video output:

- Three Digital to Analog Converters (DAC)
- DAC LUT (look-up table)
- Overlay Color Table
- Pixel Mask

Figure 7 is a block diagram which shows the flow of data within the display module. The 8 bit image first passes through the pixel mask which allows the programmer to strip bits from each pixel. This could be used to limit the number of values used for each shade of gray and therefore, certain thresholds of differences between grayscales. For example, if only two values were required for further image processing, the pixel mask could be set to 1000 0000 binary. This would map the grayscale input to either 00 or 80 hexadecimal and any pixel with a grayscale of 127 or less would be displayed as white and any pixel with a value greater than 127 would become gray.

After exiting the pixel mask, the pixel data is transformed into a pseudocolor image with the DAC LUT. The DAC LUT consists of red, green and blue triplet bytes that contain the conversion value for each color. Because we want to see a grey scale representation of the image, we have coded the DAC LUT to echo the value of the pixel for all three colors. The DAC LUTs could be used to accent a certain value, perhaps a threshold value of interest. In that case when a pixel with the threshold value was received, the green and blue DAC LUTs could output a 0x00 will the red outputs 0xff. This would cause all pixels with the value of interest to be displayed in red.

The final processing that takes place prior to the image being displayed is combining an overlay with the image. The overlay memory consists of 1024x1024x4 bits

of data. When the value of the 4 bits is zero, no overlay is displayed. The other 15 values can be obtained from the 4 bits and will override the image data with colors from a overlay color table. This table is accessed the same way the DAC LUT does with red, green and blue data obtained for each pixel.

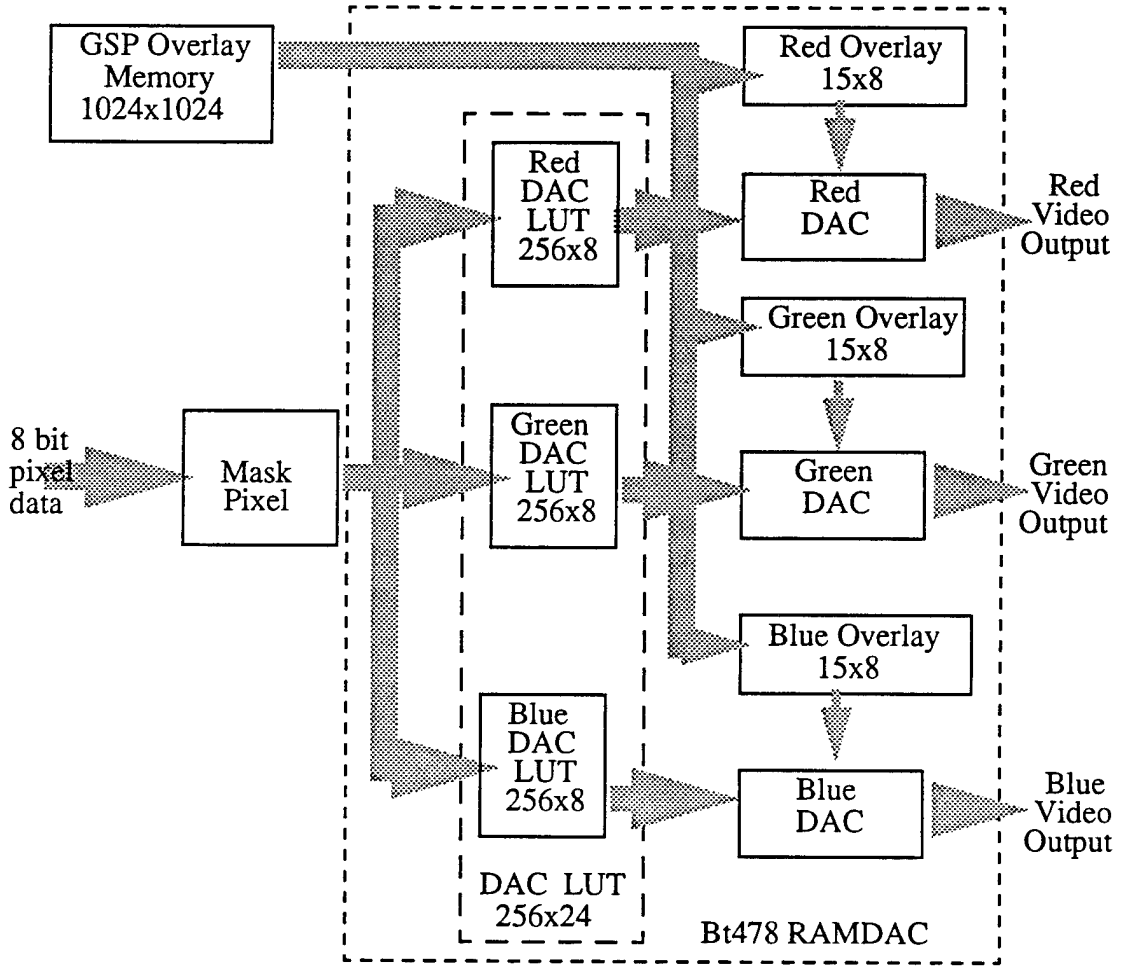

Figure 7: Display Module Analog Conversion Block Diagram

3. Display Module Registers

The display module registers are unique in that all the registers are in turn mapped to the GSP registers. There are only 16 DM registers, with 8 registers allocated for GSP

interface and 8 allocated for module identification and software reset functions. The only purpose of the display module registers is for an interface between the GSP and the programmer. This interface is accomplished with a set of three registers that read and write data to the DAC LUT, DAC overlay, DAC mask and GSP control.

a. DM Register Usage

(1) **Low Address Register**: Contains the least significant 16 bits of a GSP address. This and the following register setup the address to be read by the Host Data Port Register.

(2) **High Address Register:** Sets the most significant 16 bits of the GSP address.

(3) **Host Data Port Register:** Holds the 16 bits of data that are actually exchanged between the GSP and the display module.

(4) **Control Register:** Controls communication between the host and the GSP. Part of this register sets the interrupt processing. Since the display interrupts are not needed, the interrupt capability is disabled. In future work these registers could be used to manipulate the overlays, and possibly use the TIGA^{TM} Graphical User Interface.

b. GSP Registers

The main internal GSP registers allow the programmer to select the correct control values for different monitor types. This code allows the programmer to select from several different monitors. The monitors available are RS-170, CCIR, 1024x1024 interlaced, and non-interlaced monitors including; 640x480, 800x600 and 1024x768.

(1) **GSP Reset Register:** Resets the state of the GSP.

(2) **Identification Register:** Identifies the type and hardware revision of the current hardware.

(3) **GSP Control Register:** Sets the clock frequency and timing for the display monitor.

(4) **DAC LUT Write Address:** This GSP address specifies which of the 256 LUT values is to be written. Writing a value is accomplish by writing to the following register.

(5) **DAC LUT Data:** Contains the data at the specified read address or is the address to which data is to be written to the write address.

(6) **DAC LUT Read Address:** This GSP register specifies the address from which data is to be read.

(7) **DAC Mask:** Contains a mask that is initially ANDed with the pixel data.

D. YAMABICO'S NEW CAMERA

1. Background

To facilitate the new image processing hardware, a new camera was also obtained that would be able to take advantage of the image processing capabilities of the new image board. The choice of cameras was influenced by the previous work developing *Yamabico*'s image understanding subsystem. The biggest change was the use of an 8 bit grayscale camera vice a 32 bit RGBA color image. In the RGBA format three bytes represent the three red, green and blue values of a pixel and the fourth byte represents a transparency (Alpha) value. The Alpha value is used in graphics applications and is not needed in this research.

To simplify calculations all images that were processed for edge detection were first converted into grayscale images with each of the three bytes representing red, green and blue. The RGB values were converted into 3 equal eight bit values representing the corresponding NTSC grayscale value.

The data for all pixels is stored in memory as a long one-dimensional array which, along with an image's dimensional information, which allows traversal of the image. The order that the pixels are stored is from left to right and top to bottom. Because most of the previous work required a greyscale representation to extract edges, the image was either converted to its NTSC grayscale value or the value of the green pixel was used as a pseudo-grayscale.

The choice was made to implement Yamabico's new image system with an 8 bit Charge Couple Device (CCD) camera. The camera chosen was the Cohu 4110 from Cohu Inc. [Cohu 90]. A comparison of camera capabilities is shown in Table 5.

Capability	Cohu 4110	JVC RGB
Imager	Single CCD	Single CCD
Active Pickup Area	6.4 mm by 4.8 mm	
Resolution	739 x 484 pixels	649 x 486 pixels
Video Output	digital (analog option)	analog
Focal Length	Variable (4.16 mm nominal)	25 and 50 mm

Table 5: Cohu 4110 / JVC Comparisons

The Cohu4110 is a digital output camera with a 1/2 inch format CCD image sensor. The area is 6.4 mm x 4.8 mm and 739 x 484 picture elements. The camera transfers in parallel one eight bit pixel which represents 256 shades of gray. The digital output eliminates the need for special hardware to convert the cameras analog signal into digital. This design also isolates sensitive analog circuits away from the host computer by putting them into the camera itself. A sample of an image taken with the Cohu 4110 is shown in Figure 8. The image shows lines shows lines extracted using the sobel operator [Peterson 92].

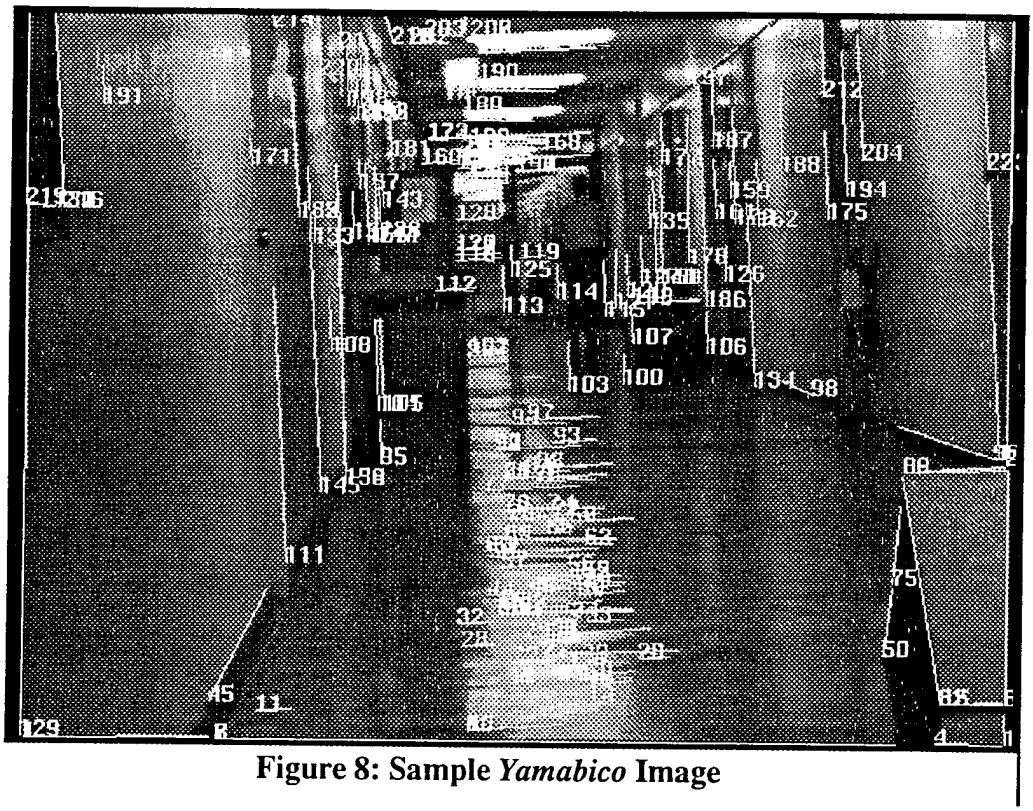

Figure 8: Sample *Yamabico* Image

III. IMAGE FUNCTIONALITY NEEDED BY MML

A. FINDING EDGES

In order to design software useful to *Yamabico*, the functionality needed for its image understanding software routines must be examined. Also, room for growth and increases in processing power must be considered.

1. Current Image Understanding Needs

The needs of currently developed software is fairly simple and well within the capabilities of the hardware. There are two basic functions needed: a snapshot capability and a method of storing the image in memory for use in a program.

Using the Personal Iris™ workstation the method has been to first grab the picture using the Silicon Graphic's video framer command "grab", which stores the image in a file using a RGBA format. Secondly, the image is retrieved from the file into a program data-structure, which contains the image header information in addition to the actual image data.

Since *Yamabico* has no file capability yet, the on-board image will only need to be stored in memory. However, the format of the image in memory should allow it to be downloaded to a workstation for further analysis.

2. Future Image Understanding Needs

It is hard to predict additional capabilities that may be needed in the future, but based on the capabilities of the camera and image grabber, three additional capabilities may be needed: zoom, hardware sobel operator, comparison of consecutive images.

A zoom capability may be useful when there is an "area of interest" within the image. The image manager has the capability to zoom in on a portion of the image. This would allow *Yamabico*'s computations to be concentrated in an area of the image which contained an obstacle or other object of interest.

Currently *Yamabico* implements the sobel operator in software, but this function is becoming prevalent in hardware. Since most of the image processing time is spent doing the sobel operator calculations. This capability should not be discounted.

The comparison of two images taken from the same viewpoint can be used to detect moving objects, by simply comparing the two images. It is for this reason the multiple frame snapshot capability should also be kept.

These three examples show that even though a capability is not currently used by *Yamabico*, that capability should not be made unavailable. The image software was written with the idea that the flexibility of the image manager hardware should not be diminished by the software. A good example of this is the three frames available to the programmer on the image manager. Current software implemented on the Personal Iris ™ workstation only has the need for one image, so the software only needs to have the capability to route images through one of the three frames; however this would limit the number of images that could be grabbed in a small period of time. The software should be designed so that an image can be directed to any one or all of the three frames.

IV. LOW LEVEL IMAGE ROUTINES

A. DISPLAY INITIALIZATION

The display initialization contains the smallest amount of code, yet it is the most complex. It follows these steps:

- •Check for correct display module present
- •Reset the display module's control register
- •Select the type of monitor desired
- •Initialize the graphics system processor (GSP)
- •Write the digital-to-analog lookup table values (DAC LUT)
- •Clear the overlay program and data memory

The first step just confirms the presence of the correct display module. It informs the user that the module is either not present or of a different model type. The next step selects the type of monitor that is to be used by *Yamabico*. The types of monitors available are listed in Table 6.The selection of monitor types is passed to the display initialization routine from the main image initialization routine which is discussed in section C. The

Type	Dimensions	Interlaced
RS-170	512x480	yes
CCIR	512x512	yes
RS-170sq	640x480	yes
CCIRsq	768x512	yes
CCIRsq	768x574	yes
1Kby1K	1024x1024	yes
--	640x480	no
--	800x600	no
--	1024x768	no

Table 6: Monitors available to *Yamabico*

procedure "selectMonitor" and the remaining routines all use three display module registers to insert values into the GSP registers. This is accomplished by loading low and high byte portions of a GSP addresses into display address registers. A final write to a data registers inserts the data at the address specified.

The initialization of the graphics system processor (GSP) takes the most time of all the initialization routines. When they have completed, the overlay program and data regions are cleared and the DAC look-up table has the correct values.

B. ACQUISITION INITIALIZATION

The acquisition module initialization routines are responsible for setting registers that control the camera/acquisition module interface. This includes:

- Checking that the correct display module is present
- Setting the clock frequency/divisor/mode
- Setting the sample edge of line, frame, field timing signals
- Setting the external synchronization parameters
- Setting the horizontal active and offset values
- Setting the vertical active and offset values
- Enabling the acquisition trigger

These routines were the hardest to debug since manuals from two different companies were needed, and the hardware was designed to work on several types of machines.

C. STANDARD IMAGE MANAGER INITIALIZATION

The standard image manager's initialization routine first sets the paging register to allow selection of frame memory, acquisition registers or display registers. The display initialization routine is called first. It is followed by the acquisition initialization routine. The frame masks are then cleared to allow 8 bits of pixel data to pass to each frame. Two control registers are then set. The first is for the image manager, which sets the clock frequency, along with enabling display and acquisition. The second control register is

designed to set zoom values. It sets them to zero since they are not used currently by *Yamabico*. Finally the pan and scroll values are then set to zero.

D. IMAGE TRANSFER SETUP ROUTINES

With all the initialization complete only the building blocks of the "grab" and "snap" remain to be constructed. They consist of the following.

1. setInputPath

This routine selects the source from which the image should come and the frame in which it should be deposited. Multiple paths can be specified. Examples are:.

```
setInputPath(A1, A0);
setInputPath(B1, CAMERA);
setInputPath(A0, CONSTANT);
```

The first statement selects frame A0 as the input for frame A1. The second statement selects the camera image to be the input for frame B1. The last example selects input for frame A0 to be the value set in the constant register.

2. setFrameAcquire

This routine selects between the grab, snap and freeze operations to be performed on each frame. This routine only sets up the frame for the specified operation. The image operation is actually executed by the "acqEnable" command. Examples of setFrameAcquire are:

```
setFrameAcquire(A0, FREEZE);
setFrameAcquire(A1, GRAB);
setFrameAcquire(B1, SNAP);
```

The first example sets frame A0 so that it does not receive any new data. The second example sets frame A1 to continuously receive images. The last example sets frame B1 to receive a single image. All three frames can be set to GRAB, SNAP or FREEZE.

3. acqEnable

This routine initiates the operation specified by the previous three operations. Prior to this operation, the frames are waiting for a grab or snap operation. Each frame is ready

to receive the image or constant data. This command starts the acquisition. The command is the parameterless function call: acqEnable();

E. GRAB, SNAP AND FREEZE

Combining the operations from the previous section, allows the formation of the primitives needed to capture an image. Here are some examples.

1. Continuously Put Images into Frames A0, A1 &B1

```
setInputPath(A1, CAMERA);
setInputPath(B1, CAMERA);
setInputPath(A0, CAMERA);

setFrameAcquire(A0, GRAB);
setFrameAcquire(A1, GRAB);
setFrameAcquire(B1, GRAB);

acqEnable();
```

2. Stop Putting Images into Frames A0, A1 & B1

```
setInputPath(A1, CAMERA);
setInputPath(B1, CAMERA);
setInputPath(A0, CAMERA);

setFrameAcquire(A0, FREEZE);
setFrameAcquire(A1, FREEZE);
setFrameAcquire(B1, FREEZE);
```

3. Snap a Single Image

```
setInputPath(A1, CAMERA);
setInputPath(B1, CAMERA);
setInputPath(A0, CAMERA);

setFrameAcquire(A0, SNAP);
setFrameAcquire(A1, SNAP);
setFrameAcquire(B1, SNAP);

acqEnable();
```

4. Set frames A0, A1 and B1 to a Constant Value

```
setInputPath(A1, CONSTANT);
setInputPath(B1, CONSTANT);
setInputPath(A0, CONSTANT);

setFrameAcquire(A0, SNAP);
setFrameAcquire(A1, SNAP);
setFrameAcquire(B1, SNAP);

acqEnable();
```

F. SUPPORT ROUTINES

Two additional support routines were developed to help in the development of grab, snap and freeze.

1. setFirstKToConstant

This routine clears the entire frame memory in which the image exists. This is needed because the image is inserted in memory at every 1024 pixel boundary. While developing the image software, this routine was used to clear the unused sides of the display of garbage. Setting the input path to a constant and snapping clears only the memory in which the pixels exist. The routine traverses the entire 1 Megabyte region setting each pixel to the constant.

2. imageTest

In order to isolate bugs in the image routines from others that may exist in MML a test routine was developed that only used MML's VME bus and standard I/O functionality This was a stand alone program that could also be used to test further image functionality, without encountering complications caused by MML, sonar or motion control interrupts. The test routine is shown in Appendix A.

V. ADAPTING SOFTWARE FOR AUTONOMOUS USE

Since the image subsystem must work with MML, the placement of sub-routine calls within MML must be carefully considered. *Yamabico's* motion control and sonar have interrupt driven routines. For this reason, all image initialization subroutines take place prior to both sonar and motion control initialization.

Once the image manager and support modules are initialized, the capturing of an image is independent of other code that is running. This is important because the motion control along with the sonar code is interrupt driven. There are four levels of interrupts in MML which correspond to hardware reset, motion control, sonar control and the user code. The current image software does not need an interrupt level since, once a grab or snap is initiated in the user code, all the remaining processing is done on the image grabber.

MML has two types of commands: sequential and immediate. This creates another challenge for the image system. In MML two commonly used statements are line() and bline(). Normally if two consecutive bline() statements were present in the user program, the first line would be completed prior to the second as shown in Figure 9. These are sequential commands. MML puts these statements into a queue so that they can be executed sequentially. MML uses this to create smooth transitions between lines. In a program with the statement, bline() followed by another bline(), the motion control subsystem would execute the two statements in order as shown in Figure 9.

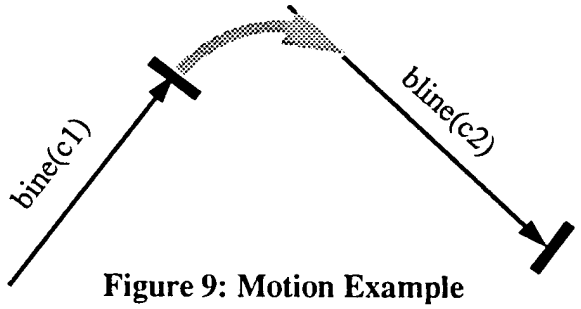

Figure 9: Motion Example

The second type of control statements are called immediate statements. These happen as they are encountered in the user program. An example is the Sonar() statement. The following code segment demonstrates the use of the immediate command Sonar().

```
void user2()
{
 int GOING = 1;
 double hit;
 CONFIGURATION  Start, cp, Jump, NewPath;

 Start = defineConfig(0.0, 0.0, 0.0, 0.0);
 Jump = defineConfig(0.0, 25.0,-1.5*HPI, 0.0);
 EnableSonar(S000);
 setLinVelImm(20.0);
 setSigmaImm(10.0);
 setRobotConfigImm(Start);
 hit = 9999.9;
 line(Start);
 do{
     while(hit >=100.0 || hit <= 1.0)
       {
        hit = Sonar(S000);
        waitMS(30);
       }
     cp = getRobotConfig();
     NewPath = compose(&cp, &Jump);
     setRobotConfigImm(NewPath);
     waitSec(3);
     hit = Sonar(S000);
  }while(GOING);
}
```

This user program sends the robot on a straight line until an object is encountered. The robot then makes a right turn of 135 degrees. The Sonar() command returns the range in centimeters from the object. The inner "while loop" is needed to keep the user program from executing the last portion of the outer "while loop". It keeps the robot on a straight line until an object is encountered at a distance of 100 cm or less.

The image subsystem has a similar problem which must be considered. The following code was designed to have the robot transition from one line to another prior to capturing a single image.

```
bline(config1);
bline(config2);

setInputPath(B1, CAMERA);
setFrameAcquire(B1, SNAP);
acqEnable();
```

However, the program does not do this. When the program executes, the two bline statements are placed in a queue and the first one is taken for execution by motion control at the next motion control cycle (every 10msec). But the user program does not stop at this point and when it continues, the three image statements are executed resulting in the image being captured during the initial portion of the first bline() statement.

A simple solution to this problem is to make the user program wait to execute its code using the MML waitMotionEnd() statement. This statement prevents the execution of any remaining statements until the queue for sequential statements is empty. The following corrected code executes as was originally intended.

```
bline(config1);
bline(config2);

waitMotionEnd();    /* Do not start taking pictures until turn is complete */
setInputPath(B1, CAMERA);
setFrameAcquire(B1, GRAB);
acqEnable();
```

VI. INTEGRATION OF IMAGE UNDERSTANDING SOFTWARE WITH MML

A. USER FUNCTIONS

The integration of an image subsystem into the motion control and sonar code could have resulted in several changes to MML. This was avoided to permit development and testing of both the image routines and MML routines separately. This allows the programmer of either set of routines to isolate errors more effectively, since the possible sources of errors is limited to the respective subsystem. The MML and the image subsystems could be combined later to test the interaction between them.

The result of these objectives was minimal impact of the image subsystem on the operation of any other subsystem. Only two changes were made to the original MML to accommodate the image subsystem. The first change was to include a single header file in the main program of MML. Secondly, the image subsystem initialization routine was added to the main program. This file is shown in Appendix B. After this code was changed, image commands could be added to the normal user.c subroutine. An example is shown below.

```
int
user()
{/* jck test program for grabbing/snapping an image  while in motion*/
  CONFIGURATION config1, config2, config3, config4;

  setInputPath(B1, CAMERA);
  setFrameAcquire(B1, GRAB);
  acqEnable();                          /* Start taking pictures */

  config1 =     defineConfig(  0.0, 60.0,-PI/2.0,  0.0);
  config2 =     defineConfig( -100.0,-30.0,-PI,    0.0);
  config3 =     defineConfig( -100.0,-30.0, 0.0,   0.0);
  config4 =     defineConfig(  0.0, 60.0,  PI/2.0, 0.0);

  setRobotConfig(config1);     /* Go out a ways and take a right turn */
  line(config1);
  bline(config2);
  Rotate(-PI);

  waitMotionEnd();              /* not until rotate done */
  setFrameAcquire(B1, FREEZE);     /* Stop taking pictures */
  setRobotConfig(config3);         /* These send the robot */
  line(config3);                /*  back to starting  */
```

```
    bline(config4);              /* position */

    waitMotionEnd();    /* Do not start taking pictures until back home */
    setInputPath(B1, CAMERA);
    setFrameAcquire(B1, GRAB);
    acqEnable();


    Rotate(PI);
}
```

The motion that results from this program is shown in Figure 10. The robot is continually grabbing images from the start of its motion until it completes the Rotate(-PI) command. Frame B1 is then frozen and does not start acquiring images until it reaches the point from which it started and starts to execute the Rotate(PI) command.

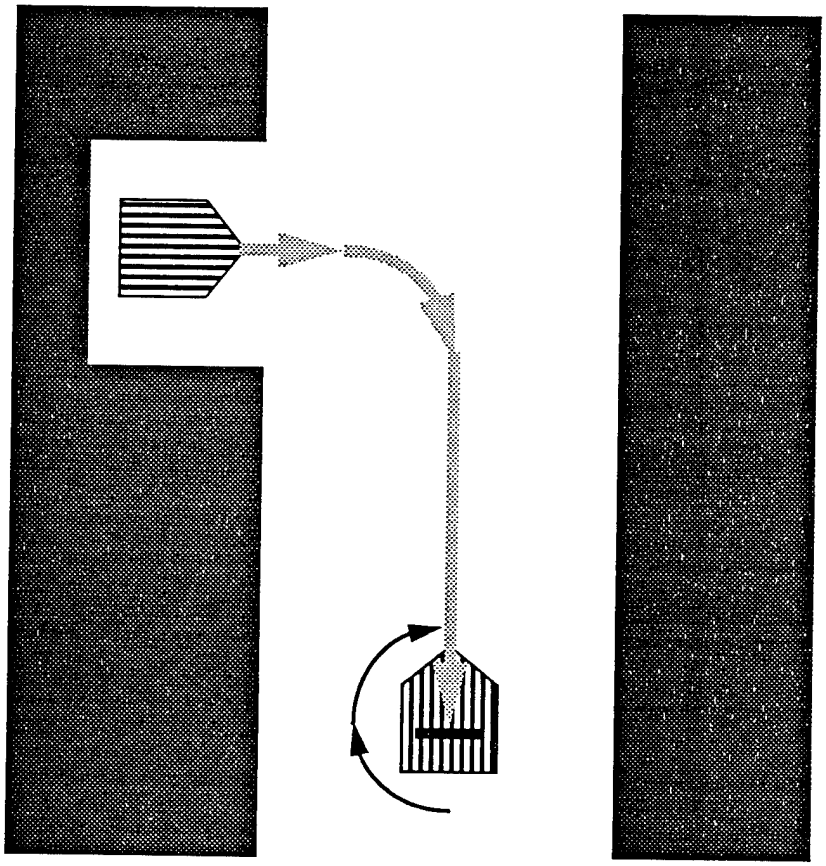

Figure 10: Diagram of Simple Robot Motion

VII. RESULTS

With the addition of this image software *Yamabico* can now grab images using a standard image manager and display those images on several types of monitors. The images can also be copied to the main memory for further processing. The image grabbing portion of *Yamabico*'s image understanding now exists autonomously on-board the robot. *Yamabico* can now obtain visual information about its environment. It can continuously grab images as it moves and display them on a monitor. It can also take a snapshot of its environment, put that image in one frame, then take another image and store that in a second frame for later comparison. All this image functionality can take place while *Yamabico* traverses its environment.

VIII. CONCLUSIONS

A. SUMMARY

The main goal of this thesis was to construct an image subsystem for *Yamabico* which exists on-board the robot. This is the first step toward the evolution of an autonomous image understanding sub-system. There were several sub-goals. One was to make the image subsystem compatible with the existing Model-based Mobile Robot Language (MML). The image system can currently operate while the robot is using its motion capabilities. Another goal that was achieved was to write all code in the C language using current software engineering guidelines. This allows the program to be modified or updated with relative ease. Lastly all the software developed here provides the functionality on-board *Yamabico* that was previously used to develop image understanding software on a Personal Iris ™ workstation. The porting of the software from those workstations onto *Yamabico* should be made easier because of this.

B. FUTURE RESEARCH

As with any research, answering a few questions only spawns several new questions. *Yamabico* is an excellent research tool to find the answers to several questions.

Can the on-board image subsystem be expanded to include image understanding subroutines developed by previous students? There are several image understanding programs that have been developed on a Personal Iris™ workstation. These routines include software to find edges in an image and match those edges to those in a model of the environment. By converting those programs for on-board use, *Yamabico*'s ability to detect and avoid obstacles could be greatly increased.

Would the introduction of additional immediate and sequential commands enhance Yamabico's image understanding capabilities? Newly developed wall-following and path

planning software may allow the robot to operate independently for longer periods of time. Perhaps vision based obstacle detection and avoidance planning should be embedded into MML. An interrupt driven image understanding routine could continually look for obstacles independent of the other subsystems.

Can *Yamabico*'s two sensors, sonar and vision, communicate with each other to provide better obstacle detection, position determination or object recognition? As discussed earlier, each sensor has its strong points. The fusion of these two sensors could provide *Yamabico* with information greater than that of each alone.

Can the analysis of color variations in an image provide *Yamabico* with additional information of its environment? Although line extraction is computationally easier using grayscales, the additional information contained in a color image may be helpful in finding edges. The detection of a change in a surfaces RGB value may allow *Yamabico* to find more edges.

Can *Yamabico* be made available to other users, such as other military or civilian schools? With the increased availability of the internet, students other than those at NPS could use Yamabico for testing of new algorithms for image understanding, sonar interpretation or motion control. Since all of Yamabico's software is written on Sun SPARC workstations and compiled using GNU's gcc, there should be a long list of schools that could use File Transfer Protocol (FTP) to obtain our source code. The code could then be altered and run on Yamabico using FTP and telnet. Perhaps even the Multicast Backbone (MBone) [Macedonia 94] could be used to monitor the execution of the test program, and logging data retrieved via FTP.

APPENDIX A. MAIN ROUTINE

This appendix contains the main test routine for the image subsytem by itself.

```
/*************************************************************
FILENAME:       imageTest.c
DESCRIPTION:    This file contains image test routines
REVISION HISTORY:  jck: LT John Kisor USN Winter 95'
jck 950110
*************************************************************/

/*****************************
jck 950111
To shorten the descriptions below several abbreviations are used:
AM: Acquisition module
DM: Display module

A0: Frame A0
A1: Frame A1
B1: Frame B1
Acq: acquisition (as in waitAcq, waitAquisition)
IMS Ref Man: Imaging Technology Inc's IMS (Standard Image Manager)
        reference Manual for series 150/40
*****************************/

#include "definitions.h"
#include "system.h"
#include "stdiosys.h"

#include "displayCtrl.h"
#include "acquisitionCtrl.h"
#include "imsControl.h"
#include "imageTest.h"

/* jck950110 This is here for testing purposes. It will be removed when
** integrated into mml
*/

void __main(void);
void Unexpected(void);


/
*************************************************************************
************
 Routine ___main is required when using the 'gcc' compiler. This is because the
compiler inserts a call to this routine at the beginning of the main function
 defined for the program. This is normally taken care of by linking in the
 bootstrap object modules, however these are not added to a program that
 operates without an operating system such as the mml program. Therefore, since
this routine is called, the only requirement is for this routine to simply
 return back to the main program.
*************************************************************************
************/
void __main()
```

```
{ /* empty */ }

/
**********************************************************************
**********Function Unexpected is the C version of Scott's blank interrupt handler
**********************************************************************
**********/void Unexpected(void)
{ /* empty */ }

/**** Local Prototypes *****/

int
getChoice(void);
/* Gets the menu choice that is desired
*/

void
interlacePage2(void);
/* Copies the page 2 part of an interlaced image into between the lines
** of page one of the interlaced image to make a truely interlaced picture
*/


int
main()
{/* jck test program for grabbing/snapping an image */

BOOLEAN readyToExit = FALSE; /* The exit flag */

  puts("Starting Main");
  InitCPU();
  InitHardware();

  while (!readyToExit)
    switch (getChoice()) {
      case 1: /* Start grabbing images */
        setInputPath(A1, CAMERA);
        setInputPath(B1, CAMERA);
        setInputPath(A0, CAMERA);
        puts("Input paths set: Camera->A0, Camera->A1 and CAMERA->B1");

        setFrameAcquire(A0, GRAB);
        setFrameAcquire(A1, GRAB);
        setFrameAcquire(B1, GRAB);
        puts("Set up to grab images into frames A0, A1 and B1");

        cycleThroughLEDs();
        acqEnable();
        puts("Frame has been grabbed");
        break;

      case 2: /* Stop grabbing images */
        setInputPath(A1, CAMERA);
        setInputPath(B1, CAMERA);
        setInputPath(A0, CAMERA);
        puts("Input paths set: Camera->A0, Camera->A1 and CAMERA->B1");

        setFrameAcquire(A0, FREEZE);
        setFrameAcquire(A1, FREEZE);
        setFrameAcquire(B1, FREEZE);
        puts("Stop grabbing images into frames A0, A1 and B1");
```

```
    break;
   case 3: /*Snap an image */
    setInputPath(A1, CAMERA);
    setInputPath(B1, CAMERA);
    setInputPath(A0, CAMERA);
    puts("Input paths set: Camera->A0, Camera->A1 and CAMERA->B1");

    setFrameAcquire(A0, SNAP);
    setFrameAcquire(A1, SNAP);
    setFrameAcquire(B1, SNAP);
    puts("Set up to Snap an image into frames A0, A1 and B1");

    cycleThroughLEDs();
    acqEnable();
    puts("Frame has been snapped");
    break;

 case 4: /* Zeroize (clear) pixel data in all frames B1 */
    setInputPath(A1, CONSTANT);
    setInputPath(B1, CONSTANT);
    setInputPath(A0, CONSTANT);
    puts("Input paths set: Constant->A0, Constant->A1 & Constant->B1");

    setFrameAcquire(A0, SNAP);
    setFrameAcquire(A1, SNAP);
    setFrameAcquire(B1, SNAP);
    puts("Set up to Snap an image into frames A0, A1 and B1");

    cycleThroughLEDs();
    acqEnable();
    puts("Frame has been snapped");
    break;

 case 5: /* Zeroize (clear) 1 MByte frame page A0 */
    interlacePage2();
    break;


 case 6: /* Zeroize (clear) 1 MByte frame page A0 */
    setFirstKToConstant(A0, 0x0000);
    break;

 case 7: /* Zeroize (clear) 1 MByte frame page A1 */
    setFirstKToConstant(A1, 0x0000);
    break;

 case 8: /* Zeroize (clear) 1 MByte frame page B1 */
    setFirstKToConstant(B1, 0x0000);
    break;

 case 9:
    selectPage(A0);
    puts("Frame A0 selected");
    readyToExit = TRUE;
    break;

 case 10:
    selectPage(A1);
    puts("Frame A1 selected");
    readyToExit = TRUE;
    break;
```

```
        case 11:
          selectPage(B1);
          puts("Frame B1 selected");
          readyToExit = TRUE;
          break;

        case 12:
          selectPage(AM);
          puts("Acquisition module selected");
          readyToExit = TRUE;
          break;

        case 13:
          selectPage(DM);
          puts("Display module selected");
          readyToExit = TRUE;
          break;

        default:
          puts("Error:main - Illegal option selected");
          rexit();
    }
    return 0;
}


int
getChoice(void)
/* Gets the menu choice that is desired
*/ {

  puts("\n\nWhat would you like to see?");
  printf("\n Enter  1 Start grabbing images.");
  printf("\n Enter  2 Stop  grabbing images.");
  printf("\n Enter  3 Snapshot (snap an image)");
  printf("\n Enter  4 Zeroize (clear) pixel data in all frames.");
  printf("\n Enter  5 Interlace the image.");
  printf("\n Enter  6 Zeroize (clear) 1 MByte frame page A0");
  printf("\n Enter  7 Zeroize (clear) 1 MByte frame page A1");
  printf("\n Enter  8 Zeroize (clear) 1 MByte frame page B1");
  printf("\n Enter  9 Exit with frame A0 selected");
  printf("\n Enter 10 Exit with frame A1 selected");
  printf("\n Enter 11 Exit with frame B1 selected");
  printf("\n Enter 12 Exit with Acquisition Module registers selected");
  printf("\n Enter 13 Exit with Display Module registers selected");
  printf("\n\n The choice is : ");
  return( GetInt() );

}

void
interlacePage2(void)
/* Copies the page 2 part of an interlaced image into between the lines
** of page one of the interlaced image to make a truely interlaced picture
*/ {
int i, j;
unsigned long page1Index = 0xfa000000; /* First empty line for pg2 pixels */
unsigned long page2Index = 0xfa07a400; /* First 1K of pixels on page 2*/
unsigned long page1Destination = 0xfa000800;
unsigned long page2Source = 0xfa07a400;
```

```
  selectPage(B1);

  for (i=0; i<236; i++) {
printf("The indexes are now %x and %x.\n",page1Index, page2Index);
    for (j=0; j<366; j++) { /*732 bytes but 2 bytes per access */
      *(WORD*)page1Destination = *(WORD*)page2Source;
      page1Destination += 2;
      page2Source += 2;
    }
    page1Index += 0x800;
    page2Index += 0x800;
    page1Destination = page1Index;
    page2Source = page2Index;
  }


}
```

APPENDIX B. MAIN.C AND USER.C

This appendix contains the main for MML it shows the additional header declaration and the initialization call for the image subsystem. Section B shows a sample user procedure which demonstrates some of the capabilities of the new image software.

```
/
*************************************************************************
************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       December 8, 1993
  Revised:    March 4, 1994
  File Name:  main.c
  Environment: GCC ANSI C compiler for the motorola 68020 processor
  Description: This file contains main(). Its purpose is to initialize all
          sub-systems and then pass control to user(). Once user() is
          complete, the routine returns control to the resident debugger.
*************************************************************************
************/

#include "definitions.h"
#include "memsys.h"
#include "serial.h"
#include "queue.h"
#include "trace.h"
#include "stdiosys.h"
#include "motion.h"
#include "time.h"
#include "sonar.h"
#include "imsControl.h"  /* Added for image routine initiialization */
#include "sonarcard.h"
#include "system.h"


/*** Local Prototypes ***/
void user(void);
void __main(void);
void Unexpected(void);



int
main()
{
  InitCPU();       /* This _must_ always be the first function called
                   because it initializes memory regions */

  initIMS();       /* jck950226 Initialize the image subsystem */

  InitTime();

  InitQueue();     /* init instruction buffer */
```

```
    InitTrace();     /* init trace mechanism */

    InitMemsys();    /* memory manager - lites LED 5 */

    InitMotion();    /* init motion, wheels, and motion logging */

    InitSonar();

    DisableInterrupts();

      /* All functions above here must initialize variables only. They
        should not rely on any interrupt handlers, timers, etc. */

    InitHardware();       /* init interrupt handlers and HW registers */
                  /* Handles ALL hardware including motion,
                    serial and sonar */
#ifdef TIMER
     /* fineTiming is used for timing the motion control cycle */
    InitClocktick(0);
#endif

    EnableInterrupts();

    MotionOn();

    user();

    waitMotionEnd(); /* wait until motion execution is done */

    DisableAllSonar(); /* disable all sonars */

    MotionOff();

    printf("\n\nUser program terminated.\a\n\n");
    printf("\n Elapsed seconds is %d ", getSeconds());

    IOclose();    /* dump all the data files to the host */

    rexit();      /* clean-up */

    return 0;
}



/
**************************************************************************
************
 Routine ___main is required when using the 'gcc' compiler. This is because the
compiler inserts a call to this routine at the beginning of the main function
 defined for the program. This is normally taken care of by linking in the
 bootstrap object modules, however these are not added to a program that
 operates without an operating system such as the mml program. Therefore, since
this routine is called, the only requirement is for this routine to simply
 return back to the main program.
**************************************************************************
************/

void __main()
```

```
{ /* empty */ }

/
**********************************************************************
***********Function Unexpected is the C version of Scott's blank interrupt handler
**********************************************************************
***********/
void Unexpected(void)
{ /* empty */ }




/************************************************************
FILENAME:       user.c
DESCRIPTION:    This file contains image/motion test routines
REVISION HISTORY:  jck: LT John Kisor USN Winter 95'
jck 950110
************************************************************/


#include "user.h"
#include "displayCtrl.h"
#include "acquisitionCtrl.h"
#include "imsControl.h"

/**** Local Prototypes *****/

int
getChoice(void);
/* Gets the menu choice that is desired
*/

void
interlacePage2(void);
/* Copies the page 2 part of an interlaced image into between the lines
** of page one of the interlaced image to make a truely interlaced picture
*/


void
goTakePictures(void);
/* jck test procedure for grabbing/snapping an image during motion */

int
user()
{/* jck test program for grabbing/snapping an image */

BOOLEAN readyToExit = FALSE; /* The exit flag */

  DisableInterrupts(); /* Disable sonar, power supply cannot support both */

  while (!readyToExit)
    switch (getChoice()) {
      case 1: /* Start grabbing images */
        setInputPath(A1, CAMERA);
        setInputPath(B1, CAMERA);
```

```
    setInputPath(A0, CAMERA);
    puts("Input paths set: Camera->A0, Camera->A1 and CAMERA->B1");

    setFrameAcquire(A0, GRAB);
    setFrameAcquire(A1, GRAB);
    setFrameAcquire(B1, GRAB);
    puts("Set up to grab images into frames A0, A1 and B1");

    cycleThroughLEDs();
    acqEnable();
    puts("Frame has been grabbed");
    break;

  case 2: /* Stop grabbing images */
    setInputPath(A1, CAMERA);
    setInputPath(B1, CAMERA);
    setInputPath(A0, CAMERA);
    puts("Input paths set: Camera->A0, Camera->A1 and CAMERA->B1");

    setFrameAcquire(A0, FREEZE);
    setFrameAcquire(A1, FREEZE);
    setFrameAcquire(B1, FREEZE);
    puts("Stop grabbing images into frames A0, A1 and B1");
    break;

  case 3: /*Snap an image */
    setInputPath(A1, CAMERA);
    setInputPath(B1, CAMERA);
    setInputPath(A0, CAMERA);
    puts("Input paths set: Camera->A0, Camera->A1 and CAMERA->B1");

    setFrameAcquire(A0, SNAP);
    setFrameAcquire(A1, SNAP);
    setFrameAcquire(B1, SNAP);
    puts("Set up to Snap an image into frames A0, A1 and B1");

    cycleThroughLEDs();
    acqEnable();
    puts("Frame has been snapped");
    break;      case 4: /* Zeroize (clear) pixel data in all frames B1 */
    setInputPath(A1, CONSTANT);
    setInputPath(B1, CONSTANT);
    setInputPath(A0, CONSTANT);
    puts("Input paths set: Constant->A0, Constant->A1 & Constant->B1");

    setFrameAcquire(A0, SNAP);
    setFrameAcquire(A1, SNAP);
    setFrameAcquire(B1, SNAP);
    puts("Set up to Snap an image into frames A0, A1 and B1");

    cycleThroughLEDs();
    acqEnable();
    puts("Frame has been snapped");
    break;

  case 5: /* Zeroize (clear) 1 MByte frame page A0 */
    interlacePage2();
    break;


  case 6: /* Zeroize (clear) 1 MByte frame page A0 */
```

```
    setFirstKToConstant(A0, 0x0000);
    break;

  case 7: /* Zeroize (clear) 1 MByte frame page A1 */
    setFirstKToConstant(A1, 0x0000);
    break;

  case 8: /* Zeroize (clear) 1 MByte frame page B1 */
    setFirstKToConstant(B1, 0x0000);
    break;

  case 9:
    selectPage(A0);
    puts("Frame A0 selected");
    readyToExit = TRUE;
    break;

  case 10:
    selectPage(A1);
    puts("Frame A1 selected");
    readyToExit = TRUE;
    break;

  case 11:
    selectPage(B1);
    puts("Frame B1 selected");
    readyToExit = TRUE;
    break;

  case 12:
    selectPage(AM);
    puts("Acquisition module selected");
    readyToExit = TRUE;
    break;

  case 13:
    selectPage(DM);
    puts("Display module selected");
    readyToExit = TRUE;
    break;

  case 14:
    EnableInterrupts();
    Rotate(-PI/2);
    waitMotionEnd(); /* wait until motion execution is done */
    DisableInterrupts();
    break;

  case 15:
    EnableInterrupts();
    Rotate(PI/2);
    waitMotionEnd(); /* wait until motion execution is done */
    DisableInterrupts();
    break;

  case 16:
    goTakePictures();
    break;


  default:
```

```
        puts("Error:main - Illegal option selected");
        rexit();
    }
}



int
getChoice(void)
/* Gets the menu choice that is desired
*/ {

  puts("\n\nWhat would you like to see?");
  printf("\n Enter  1 Start grabbing images.");
  printf("\n Enter  2 Stop  grabbing images.");
  printf("\n Enter  3 Snapshot (snap an image)");
  printf("\n Enter  4 Zeroize (clear) pixel data in all frames.");
  printf("\n Enter  5 Interlace the image.");
  printf("\n Enter  6 Zeroize (clear) 1 MByte frame page A0");
  printf("\n Enter  7 Zeroize (clear) 1 MByte frame page A1");
  printf("\n Enter  8 Zeroize (clear) 1 MByte frame page B1");
  printf("\n Enter  9 Exit with frame A0 selected");
  printf("\n Enter 10 Exit with frame A1 selected");
  printf("\n Enter 11 Exit with frame B1 selected");
  printf("\n Enter 12 Exit with Acquisition Module registers selected");
  printf("\n Enter 13 Exit with Display Module registers selected");
  printf("\n Enter 14 To rotate the robot 90 degrees to the right");
  printf("\n Enter 15 To rotate the robot 90 degrees to the left");
  printf("\n Enter 16 Go forward (90cm), turn to right line (100cm), return");
  printf("\n\n The choice is : ");
  return( GetInt() );

}


void
interlacePage2(void)
/* Copies the page 2 part of an interlaced image into between the lines
** of page one of the interlaced image to make a truely interlaced picture
*/ {
int i, j;
unsigned long page1Index = 0xfa000000; /* First empty line for pg2 pixels */
unsigned long page2Index = 0xfa07a400; /* First 1K of pixels on page 2*/
unsigned long page1Destination = 0xfa000800;
unsigned long page2Source = 0xfa07a400;

  selectPage(B1);

  for (i=0; i<236; i++) {
    for (j=0; j<366; j++) { /*732 bytes but 2 bytes per access */
      *(WORD*)page1Destination = *(WORD*)page2Source;
      page1Destination += 2;
      page2Source += 2;
    }
    page1Index += 0x800;
    page2Index += 0x800;
    page1Destination = page1Index;
    page2Source = page2Index;
  }
```

```
}

void
goTakePictures(void)
{/* jck test procedure for grabbing/snapping an image during motion */
  CONFIGURATION config1, config2, config3, config4;

  setInputPath(B1, CAMERA);
  setFrameAcquire(B1, GRAB);
  acqEnable();

  config1 =     defineConfig(  0.0, 60.0,-PI/2.0,  0.0);

  config2 =     defineConfig( -100.0,-30.0,-PI,    0.0);/*-375 to doorway*/

  config3 =     defineConfig( -100.0,-30.0, 0.0,   0.0);

  config4 =     defineConfig(  0.0, 60.0,  PI/2.0, 0.0);

  Rotate(-PI/2);

  Rotate( PI);                /* Look both ways */

  waitMotionEnd();              /* not until rotate done */

  setFrameAcquire(B1, FREEZE);    /* Stop taking pictures */

  Rotate(-PI/2);

  setRobotConfig(config1);    /* Go out a ways and take a right turn */

  line(config1);

  bline(config2);

  Rotate(-PI);

  waitMotionEnd();   /* Do not start taking pictures turned back home */

  setInputPath(B1, CAMERA);
  setFrameAcquire(B1, GRAB);
  acqEnable();

  setRobotConfig(config3);        /* These send the robot */

  line(config3);              /*  back to starting  */

  bline(config4);              /* position  */


  Rotate(PI);
}
```

LIST OF REFERENCES

[Book 94] Book, S., "*Improving Software Characteristics of a Real-time System Using Reengineering Techniques*", Master's Thesis, Naval Postgraduate School, Monterey, California, March 1994.

[Cohu 90] Cohu, Inc., "*Installation and Operation Manual for 4110 Digital Output Monochrome CCD Camera*", Technical Manual Code 6X-899, August 1990.

[DeClue 93] Declue, M., "*Object Recognition Through Image Understanding for an Autonomous Mobile Robot*" Master's Thesis, Naval Postgraduate School, Monterey, California, March 1994.

[ITIAM 93] Imaging Technology Inc., "*AM-DIG Hardware Reference Manual*" Document Number 47-H44002-00, October 1993

[ITIDM 92] Imaging Technology Inc., "*DM-PC Hardware Reference Manual*" Document Number 47-H44001-00, September 1992

[ITIIMS 93] Imaging Technology Inc., "*IMS Hardware Reference Manual*" Document Number 47-H54002-00, April 1993

[Kanayama 94] Kanayama, Y., "Mathematical Theory of Robotics: Introduction to 2D Spatial Reasoning", *Lecture Notes of the Advanced Robotics Course*, Department of Computer Science, Naval Postgraduate School, Winter Quarter 1994.

[Lochner 94] Lochner, J., "*Analysis and Improvement of an Ultrasonic Sonar System on an Autonomous Mobile Robot*" Master's Thesis, Naval Postgraduate School, Monterey, California, March 1994.

[Macedonia 94] Macedonia, M., Brutzman, D., "*MBone Provides Audio and Video Across the Internet*" IEEE Computer, April 1994, pp1-11

[Peterson 92] Peterson, K., "*Visual Navigation for an Autonomous Mobile Vehicle*", Master's Thesis, Naval Postgraduate School, Monterey, California, March 1994.

[Stein 92] Stein J., "*Modeling, Visibility Testing and Projection of an Orthogonal Three Dimensional World in Support of a Single Camera Vision System*", Master's Thesis, Naval Postgraduate School, Monterey, California, March 1994.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center — 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Dudley Knox Library — 2
   Code 052
   Naval Postgraduate School
   Monterey, CA 93943-5101

3. Chairman, Code CS — 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

4. Dr Yutaka Kanayama, Code CS/KA — 3
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

5. Commanding Officer, VS-21 — 2
   Sea Control Squadron Two One
   FPO AP, 96601-6500
   Attn: Lt. John C Kisor